Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

ELSEVIER

Computer Networks 42 (2003) 461–479

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

A protocol-adaptive monitoring tree for efficient design of traffic monitoring probes

E. Magaña [a,b,*], J. Aracil [b], J. Villadangos [b]

[a] *Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720, USA*
[b] *Departamento de Automática y Computación, Universidad Pública de Navarra, Campus de Arrosadia, 31006 Pamplona, Spain*

Received 4 April 2001; received in revised form 22 January 2003; accepted 6 February 2003

Responsible Editor: R. Stadler

Abstract

This paper presents a novel protocol-adaptive monitoring tree (PAM-Tree) algorithm. PAM-Tree is a filtering algorithm that fulfills the needs of a general-purpose traffic monitoring system. PAM-Tree is aimed at overcoming the inefficiencies of existing stream-oriented traffic probes in traffic monitoring scenarios where large numbers of filters change dynamically. Both analytical and experimental performance evaluations show that PAM-Tree has a great potential to be an efficient filtering engine for high-speed loss-less monitoring systems.

Keywords: Traffic monitoring tools; Network performance measurements; Packet filters

1. Introduction

As network capacity continues to increase worldwide, with thousands of hosts and a wide variety of protocols and services, traffic monitoring has become an increasingly challenging task. Traffic monitoring systems [18] are composed of probes, which are hardware or software devices in charge of measuring traffic from a given network segment, as shown in Fig. 1. Such probes deliver reports and statistics to the measurement console.

While traffic parameters are specified in network monitoring standards such as RMONv2 [24], there is very little in the literature on probe architecture, implementation and performance evaluation. As a result, performance tests on commercially available RMON probes are likely to give false traffic measurements, since significant packet losses are likely to occur [21, Chapter 10.10; 22]. The reason is that traffic probes are unable to keep track of thousands of filters and associated parameters in high load situations. In fact, traffic probes impose a number of requirements that are distinct from other systems requiring packet filtering capabilities, such as routers, operating systems, firewalls and traffic analysis applications (*tcpdump* [9], for

* Corresponding author. Address: Departamento de Automática y Computación, Universidad Pública de Navarra, Campus de Arrosadia, 31006 Pamplona, Spain. Tel.: +34-948-16-9853; fax: +34-948-16-8924.

E-mail addresses: eduardo.magana@unavarra.es, emagana@eecs.berkeley.edu (E. Magaña), javier.aracil@unavarra.es (J. Aracil), jesusv@unavarra.es (J. Villadangos).

1389-1286/03/$ - see front matter
doi:10.1016/S1389-1286(03)00196-8

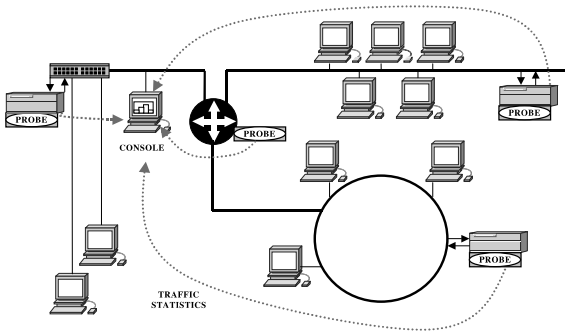

Fig. 1. Traffic measurement probes.

example). As a result, current packet filtering techniques cannot be successfully ported to the specific scenario of traffic monitoring systems.

The paper is organized as follows: the requirements imposed by traffic monitoring systems are analyzed in Section 1.1; the state of the art in packet filters and classifiers is discussed in Section 1.2; and the paper contributions are presented in Section 1.3. Section 2 is devoted to an in-depth description of the PAM-Tree (protocol-adaptive monitoring tree) algorithm. Section 3 presents analytical and experimental performance evaluations of both PAM-Tree and packet filter. Section 4 describes a PAM-Tree based monitoring system (MONET) and, finally, we present our conclusions in Section 5.

1.1. Traffic monitoring system requirements

Traffic monitoring systems can provide parameters on a wide variety of scales, from information as coarse as the total network load, to data as detailed as the WWW traffic between a pair of hosts. However, in every case, the parameter value needs to be updated as new packets appear in the network segment. Because of increasing network traffic volume, along with the large number of parameters supported by traffic probes, it turns out that the filtering engine is the key component of the monitoring system, and can possibly become a bottleneck. It is important to state that updating traffic parameters is the sole purpose of a traffic monitoring system. That fact alone makes it different from other systems with packet filtering capabilities, since it does not require a packet stream. In fact, traffic monitoring systems call for a departure from the stream-oriented paradigm, for a number of reasons.

First, a single packet may trigger the update of several parameters (a packet contributes to the total network load but may also contribute to the load between a pair of hosts). Traffic filtering algorithms usually check a single value in a certain header field. On the other hand, it is likely that filters be required to discriminate a range of values, and not a single one. Furthermore, not only may several parameters be updated with the same packet stream, but a single parameter may be affected by many packet streams. Therefore, an optimized packet filtering technique does not suffice for traffic monitoring, which demands an integrated solution that arranges filters and parameters jointly.

Second, traffic monitoring systems are required to keep track of and dynamically update a very large number of filters. Frequently, automatic filter insertion must be performed if network load grows beyond a given threshold, so that a full picture of network activity can be obtained at that time. In fact, the information provided by the monitoring system at congested times is important in detecting network weak spots. Since data traffic is inherently bursty, congestion onset may occur quickly, and response time is crucial to capturing detailed traffic statistics. While this is not an issue in cases where a single parameter is requested (e.g., the amount of traffic to a certain destination port), the same does not apply to other cases. For example, there may be a request for a traffic matrix identifying bulk traffic users. In a worst case scenario, a traffic matrix of 20 hosts implies allocating 400 filters in a very short time. We could provide additional examples in which response time is critical to maintaining valuable management information, such as the number of bytes per newly arrived TCP connections to a certain destination port, and so on.

Third, the traffic probe cannot be stopped for filter insertion since, in that case, packets for which the filter was setup could be lost. This is one of the most important distinguishing features of traffic monitoring systems.

At this point, the fact that PAM-Tree is not a generic packet filter or packet classifier must be stressed. PAM-Tree is a filtering algorithm that

fulfills the needs of a general-purpose traffic monitoring system. It is not intended to replace current implementations of packet filters, but to fill in the existing gaps in traffic monitoring.

In the next subsection, we explain why the existing systems (packet filters and classifiers) cannot fulfill the specific requirements of traffic monitoring systems.

1.2. Traffic filtering: state of the art

Aside from monolithic design approaches (which lack the flexibility required by networking scenarios suffering from constant protocol and service changes), the state of the art in traffic filtering systems can be broadly divided into two areas: (i) packet filters and (ii) packet classifiers for high speed routers.

1.2.1. Packet filters

Since traffic analysis in standard PCs and workstations is under the control of a general purpose operating system, it turns out that packet losses may occur, thus biasing the traffic analysis results. Such measurement errors are normally due to the processing burdens that result from promiscuous packet analysis at the user level, especially if network load is significant. In order to avoid such burdens, packet filters [16] operate at the kernel level, directly on top of the network interface device driver. Such packet filters serve to filter out those packets that are not relevant to the traffic analysis being performed at the application layer. As a result, the traffic monitoring server (Fig. 2) receives a refined packet stream that is easier to handle, and uses it to update the traffic parameters. We call this type of design approach *stream-oriented*.

The most commonly used packet filter is the Berkeley Packet Filter (BPF, 1993) [15]. A high-level imperative language for filter definition is provided, allowing for flexibility in defining complex filter expressions. However, this high-level language requires interpretation and subsequent compilation into the filtering engine machine code. Thus, filter insertion and removal take a long processing time. Also, since BPF is not optimized for filter reuse, redundant comparisons are likely

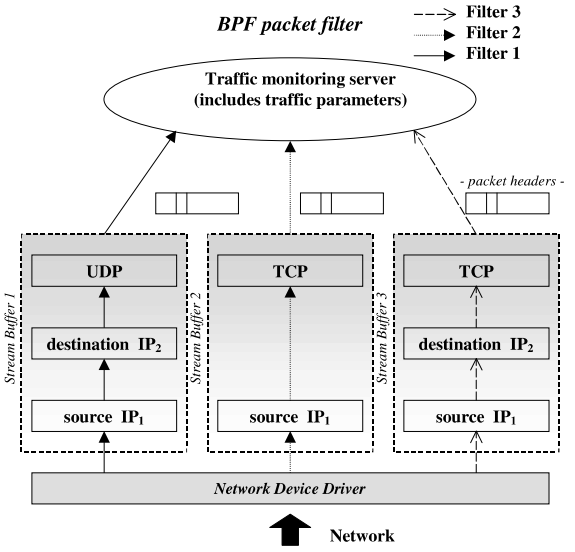

Fig. 2. Stream-oriented architecture.

to occur, resulting in CPU overhead. Earlier and concurrent packet filter implementations, such as the CSPF (CMU/Stanford Packet Filter) [16], Solaris DLPI [23] and Irix Snoop [19], have the same drawbacks and have been reported to be less efficient [15].

Match Packet Filter (MPF, 1994) [26], is the first packet filter to introduce the concept of filter block reuse, thus outperforming BPF. Additional MPF enhancements include support of a large number of filters and packet fragmentation. Nevertheless, only two filtering levels (protocol and port) are allowed, and lookup latency is reduced by means of a hash table. Consequently, filter insertion time is increased in comparison to BPF, since filter insertion requires adding new entries to the hash table.

PathFinder (1994) [2] is the first packet filter that provides a declarative filter definition language. Pathfinder's dynamic acyclic graph (DAG) allows for filter block reuse (longest prefix match) and it is also amenable to hardware implementation. Furthermore, it evaluates not only the lookup latency, but also the insertion cost. However, the inability to perform filter insertion and removal with no service interruption is a major drawback for traffic monitoring, as is the fact that Pathfinder's insertion cost is high in comparison to MPF. This is so because already existing filters in

the DAG may serve as prefixes for new filters, meaning that a prefix search is required, along with memory allocation for the new filter blocks (cells). In sum, Pathfinder is optimized for lookup cost, but not for dynamic insertion and removal.

Dynamic Packet Filter (DPF, 1996) [5], is a packet filter that exploits the flexibility of dynamic code generation, but at the expense of an even higher insertion cost than Pathfinder. Finally, a second version of BPF called BPF+ (1999) [3] incorporates an optimized packet filtering scheme that serves to reduce redundancy. However, BPF+ is also interpreted and the insertion cost is not negligible. Additionally, an input/output descriptor is required per filter [1] and, due to the operating system overhead (device creation), the insertion cost grows even higher. Nevertheless, both BPF+ and DPF are extremely efficient in providing compilation of a high-level filter definition into (virtual) machine code. Yet, while the results are impressive, such features are not required by traffic monitoring systems, which demand agile handling of a large number of filters and parameters dynamically.

Fig. 2 presents an example of a stream-oriented architecture, showing a traffic monitoring server that updates parameters according to the packet streams it receives (one packet stream per filter). A stream-oriented architecture operates as follows:

(1) Filters are defined independently and a separate queue for each filter is set up in kernel memory (a separate device in case of BPF [15]).
(2) Each incoming packet is tested against all filters.
(3) When a packet verifies a particular filter, either the entire packet or the packet header is copied to the traffic monitoring server in user space.
(4) When a packet verifies more than one filter, it is copied to the queues associated with these filters.

[1] BPF requires a kernel device */dev/bpfxxx* for each filter, thus limiting the maximum number of simultaneous filters to 256, which is the maximum value for device minor number in UNIX file systems. In order to increase the maximum number of filters, the BPF kernel module has to be duplicated.

The stream-oriented packet filter paradigm is not meant to carry out traffic monitoring functions. First, and regardless of any possible optimization methods (say, for instance, filter block reuse), the filtered packet streams are delivered from the packet filter, with traffic parameters becoming updated only at the traffic monitoring server, on top of the packet filter. In this way, packet filtering and parameter updating are unnecessarily decoupled, and the creation of packet copies results in high CPU cost. Furthermore, recall that several traffic parameters may be updated with the very same stream or, conversely, a single stream may serve to update several parameters. Therefore, either a filter for each of the parameters is defined, which will surely lead to unnecessary packet copies, or a packet classifier is required on top of the packet filter in order to update the traffic parameters.

Second, and most important, per filter buffers must be allocated/deallocated for filter insertion/removal. This is a simple requirement of a packet filter architecture (Fig. 2), regardless of the filtering algorithm used. Thus, filter insertion and removal costs are quite significant.

In conclusion, while packet filters are adequate for detailed packet stream analysis, and the contribution of packet filters to the design of traffic analysis applications has been extraordinary, the concept cannot be extended easily to traffic monitoring, since the focus in this case is no longer on the packet stream itself, but on the traffic parameters.

1.2.2. Packet classifiers for high-speed routers

The objective of packet classifiers is to minimize lookup latency for *single* routing table entries. This problem can be resolved as the point location problem in multidimensional space [7,10,20] or as the lowest cost matching-filter problem [20]. Since a packet classifier cannot be used to check several conditions, it has little applicability to the traffic-monitoring scenario, in which a single packet may be called upon to update several traffic parameters. Furthermore, since packet classifiers work on a longest-prefix match basis [6] (destination address field), they lack the filter definition flexibility needed by a traffic monitoring system. For the latter,

filters may comprise several packet header fields, not necessarily contiguous. Finally, and most importantly, packet classifiers perform filter insertion and removal only when a routing table entry is updated. Clearly, the rate for such updates is significantly less than the filter insertion and removal rate needed by a traffic monitoring system. Consequently, packet classifiers lack the dynamic features required for traffic monitoring. In fact, packet classifiers in high speed routers are optimized to minimize lookup latency [7,10,20], not to be flexible. With reconfigurations taking place every time a new filter is updated, the filter insertion processing time is too slow for traffic monitoring purposes.

1.3. This work

As an alternative, this paper presents a traffic filtering scheme called protocol-adaptive monitoring tree (PAM-Tree). PAM-Tree is based on a tree structure that can be configured according to the protocol architecture of the network being measured. PAM-Tree provides a filtering data structure that jointly incorporates parameters and filters, thus boosting the parameter update process. Consistent with the requirements of traffic monitoring systems, PAM-Tree is based on a declarative filter definition interface [1] that enables agile filter insertion and removal. PAM-Tree is the only filtering algorithm that features filter insertion and removal with no service interruption in the traffic probe, i.e., while performing traffic monitoring. The costs of both filter insertion and removal are reduced in comparison with the state-of-the-art reported figures, as we will show in the experimental analysis section. In addition, PAM-Tree is the only algorithm that has been formalized, using input/output automata [11], and the algorithm properties have been proved. Analytical performance evaluation (complexity evaluation) has been carried out, also as a distinctive feature of this work. Finally, the experimental analysis includes parameters such as packet size, network load, number of filters and filter insertion rate while in service, which have not been considered before. To the best of our knowledge, this is the first comparative analysis of existing packet filtering solutions and the first proposal of a novel algorithm for the specific field of traffic monitoring systems, which is gaining an ever-increasing importance in the current Internet environment.

2. Protocol-adaptive monitoring tree

Contrary to state-of-the-art solutions, PAM-Tree is tailored to the needs of a general purpose traffic probe. The traffic-monitoring server (Fig. 3) does not receive a packet stream resulting from application of a filter. Instead, this server submits the filter, a parameter, and a routine for traffic parameter handling to PAM-Tree. Upon receiving a packet, PAM-Tree invokes the routine, and the parameter value is updated. For most RMONv2 traffic parameters, the routine simply accounts for the number of bytes per packet or number of packet arrivals. A traffic probe architecture based on the PAM-Tree is illustrated in Fig. 3.

The PAM-Tree algorithm operates as follows: Packets are read promiscuously from the network interface card, and a copy is buffered. Then, packets are filtered using a filtering engine (*PAM-Tree graph*, which will be defined in Section 2.2) and the corresponding parameters are updated. A parameter is updated if and only if the

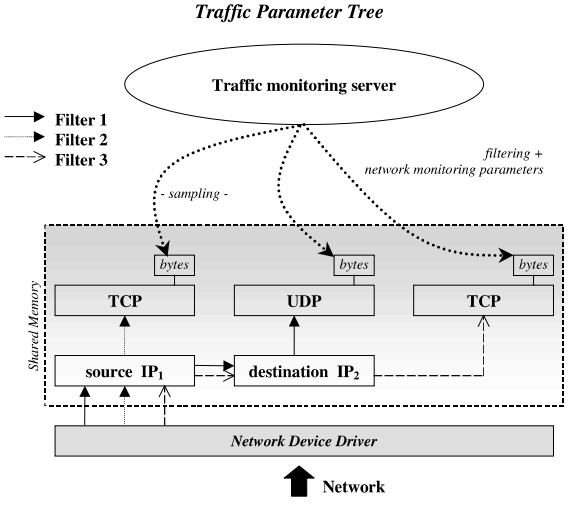

Fig. 3. PAM-Tree traffic probe architecture.

corresponding filter is verified. Recall that a packet may actually verify several filters and, thus, several parameters may be updated. Finally, the parameter value is sampled by the traffic monitoring server, with the desired sampling interval, and sent to the monitoring console.

The advantages of PAM-Tree are manifold: First, unnecessary packet copies from the filtering engine to the traffic monitoring server are avoided. Second, PAM-Tree offers the same modularity and portability as stream-oriented architectures, since the library interface is very much the same. Third, the process of setting up and/or tearing down filters is significantly faster. Filters can be inserted and removed *dynamically*, since there is no need to allocate separate memory streams, but only new parameters in the filtering engine. Finally, the packet filtering and parameter update engine is also optimized for efficiency, avoiding filter replications.

Thus, PAM-Tree can be seen as a natural and more efficient extension of the packet filter, one that provides the performance required in traffic monitoring systems. In order to optimize traffic filtering, the PAM-Tree algorithm maps a traffic parameter into a number of subfilters according to different protocol layers. For instance, a traffic parameter "number of HTTP packets between two Ethernet hosts" would produce a subfilter at the MAC layer (Ethertype equals IP), plus another three subfilters at the IP layer (Source IP address, Destination IP address and protocol selector equal to 6-TCP) [8], along with another one at the transport layer (destination port equal to 80-WWW). Should a new traffic measurement be requested, involving the same IP addresses and a different transport protocol and service, the subfilters ethertype, source and destination IP addresses could still be reused. On the other hand, PAM-Tree is not coupled to a specific protocol stack. A protocol database in the console defines the binding between the protocol stack of the network being measured and the subfilter arrangement in the PAM-Tree, through a user-friendly graphical interface.

A detailed formalized description of the algorithm internals is presented in the next section and in the Appendix A. For brevity, we do not provide the entire formal specification here, but it is available upon request [12].

2.1. Preliminary definitions

In order to define the state of the system, we first provide a set of definitions which will be used along the next sections. A packet P is defined as a finite concatenation of bits. The function $\text{length}(P)$ gives the packet length in bits. The PAM-Tree algorithm will check sequences of contiguous bits within the packet, which are referred to as fields, in order to update the corresponding parameters. For example, a field could be an IP address or a source port. Fields can be obtained by means of a mask applied to a packet. A filter is a set of subfilters, each subfilter being defined by a field and a comparison value. Finally, every filter has an associated *traffic parameter*, which is updated whenever a packet verifies the corresponding filter. It must be noted that the main purpose of PAM-Tree is to update parameters as new packets are received by the traffic probe. The above definitions are formalized as follows:

Definition 1 (*Packet*). A packet is a finite and ordered list of c_i bits, namely, $P = \bullet c_k$: $k = 1, \ldots, \text{length}(P)$, where $\bullet$ denotes the string concatenation operation.

Definition 2 (*Field*). Sequences of consecutive bits within a packet are called fields. Let $\text{fields}(P)$ be the set of fields in P and $\#\text{fields}(P)$ its cardinal. Let $I_s(j,P)$ and $I_e(j,P)$, $1 \leqslant j \leqslant \#\text{fields}(P)$ be the function that provides the position of the starting and ending bit of a field j, with $I_s(1,P) = 1$.

Definition 3 (*Field l subfilter*). Let us consider a packet P with $\text{length}(P) = n$ and $\#\text{fields}(P) = k$ fields. A field l subfilter is defined as $F_l(P) = (I_s(l,P), I_e(l,P), a, f, \Lambda)$, $1 \leqslant l < k$ and $F_k(P) = (I_s(k,P), n, a, f, \Lambda)$ where a is a real value, f is a real-valued function (intermediate processing function) that is applied to the packet field and Λ is a logical operator $(=, <, >, !=)$. Then $F_l(P) = \text{True} \iff (f(I_s(l,P), I_e(l,P))\, \Lambda\, a) = \mathit{True}$.

For each incoming packet P, a traffic parameter will be updated if and only if the packet fields satisfy a filter. A filter is defined as a combination of OR subfilter components. In what follows, "$\wedge$" represent the AND logic operation and "$\vee$" the OR logic operation.

Definition 4 (*OR component*). An OR component is defined as a concatenation of subfilters using ANDs relations: $F_{\mathrm{OR}_j}(P) = \bigwedge_{i\in\mathrm{fields}(P)} F_i(P)$.

Definition 5 (*Filter*). A filter is defined as an OR concatenation of one or more OR components: $F(P) = F_{\mathrm{OR}_1}(P) \vee F_{\mathrm{OR}_2}(P) \vee \cdots \vee F_{\mathrm{OR}_m}(P)$.

Due to the associative and distributive properties of the logic operators, any given filter can be written as in the previous expression. For example, the following filter G:

$$G = (\mathrm{IP} \wedge (\mathrm{TCP} \vee \mathrm{UDP})) \vee (\mathrm{IP} \wedge \mathrm{IP\,source})$$

can be converted to

$$G = ((\mathrm{IP} \wedge \mathrm{TCP}) \vee (\mathrm{IP} \wedge \mathrm{UDP})) \vee (\mathrm{IP} \wedge \mathrm{IP\,source})$$

Finally, the *traffic parameter* is defined as the variable that is updated whenever a packet verifies a filter.

Definition 6 (*Traffic parameter*). A traffic parameter is defined as $Q = (\mathrm{id}, F, \mathrm{UpdateFunction})$ where id is an identifier, F is a filter and UpdateFunction is a function that determines how the parameter counter must be updated when the associated filter is verified: i.e., by count of bits, packets or other.

2.2. Algorithm description

PAM-Tree is based on a data structure that allows the algorithm to insert, remove, poll and update parameters (*PAM-Tree graph*). While a pseudocode specification is provided in the Appendix A, a natural language description is given in this section for the sake of readability.

PAM-Tree is an algorithm with two main functionalities: (i) building the filtering data structure, which is denoted by *PAM-Tree graph* and (ii) updating traffic parameters. In order to update traffic parameters, incoming packets are filtered through the PAM-Tree graph. The PAM-Tree graph is a rooted directed graph, $G = (N, A)$, where N is a set of subfilter nodes (vertices), each of them representing a field l subfilter, and A is a set of pairs of distinct nodes from N, (n_i, n_j). Each pair of nodes in A is called an arc (edge). A filter F can be viewed as a subtree of G, namely, a sequence of nodes $(r, \ldots, n_m)$ such that the pairs $(r, n_2), (n_2, n_3), \ldots, (n_{m-1}, n_m)$ are arcs of G and r is the root. The subfilter node n_m incorporates the *traffic parameter* (see Definition 6) to be updated if a packet passes the filter F. A PAM-Tree graph is a dynamic structure, since new filters may be requested at any time. Reuse of common subfilter nodes takes place whenever a new filter is incorporated. An example of a PAM-Tree graph is provided in Fig. 4.

The PAM-Tree algorithm contains three functional components:

(1) *Filter expansion*: Each new parameter is expanded into a number of subfilters corresponding to the different packet fields to be tested. Such expansion is performed by means of the protocol database, in which the mapping between the parameters and subfilters is defined in advance by the network manager. This simplifies support of new protocols. An entry in the protocol database defines the

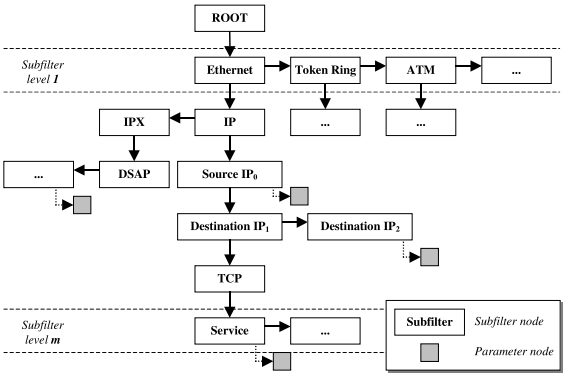

Fig. 4. PAM-Tree data structure.

binding between a traffic parameter and a number of packet fields, which are defined by relative position, size and meaning. For example, a traffic parameter "Number of IP packets between two Ethernet hosts" is expanded into three packet fields: MAC layer protocol selector field (IP), IP source address and IP destination address. These fields have a one-to-one mapping to subfilter nodes that compose the filter associated with the parameter.

(2) *Dependencies resolution*: Once the set of subfilters for each filter is available, the algorithm proceeds to check dependencies in order to optimally place the subfilters. First, subfilters are sorted in descending restrictive order. Then a recursive function is called to check if each of the subfilters has already been placed in the filtering tree: if so, no further action is performed; otherwise, the new subfilters are interleaved with existing subfilters, from most to least restrictive.

(3) *Search*: Upon a packet arrival, there is a search for matching filters, starting from the root node. In order to avoid creating packet copies, a pointer to the incoming packet, and not the packet itself, is used. The search process results in either an update of the corresponding parameter(s), or packet discard.

The proposed algorithm produces a number of desirable characteristics for a traffic monitoring system:

- *Reusability*: Since testing a subfilter at protocol layer N (IP) involves testing other subfilters at lower levels (MAC), the chance that a subfilter is reused is significant.
- *Self-organization*: Since the tree grows according to the filter expansion defined in the protocol database, the algorithm is self-organizing. Each of the traffic probes deployed in the network may be running a different filtering tree structure, depending on the protocols being observed in each particular network segment and incorporated to the database. Thus, the PAM-Tree constitutes a flexible algorithm that can be configured according to network characteristics and user preferences.
- *General purpose application*: Traffic parameters are expanded into a number of subfilters, which are defined in a protocol database. The use of a protocol database decouples parameter semantics from packet filtering, ensuring adaptability to protocols and services. In the PAM-Tree implementation, the protocol database is incorporated into the console, or, optionally, into any SQL server present in the network. In order to add a new protocol or service we simply need to incorporate it into the protocol database.

Furthermore, the following two lemmas, which are given without proof (see [12]), assert important properties of the algorithm. The first one shows that filters and parameters are inserted on-the-fly in the PAM-Tree graph. The second one shows that filters share common prefixes (subfilters), thus avoiding repeated testing of the same subfilter and greatly improving filtering performance.

Lemma 1. *The algorithm allows for filter insertion and removal with no service interruption.*

Lemma 2. *All the inserted filters reuse the longest common prefix of subfilters.*

Moreover, the algorithm allows for running several operations in parallel on the PAM-Tree graph. In this way, the algorithm can safely insert, delete, poll and update parameters, meaning that the operations can be parallelized with no risk to information consistency [12]. Thus, multiprocessor systems can be safely used to maximize algorithm performance.

3. Performance evaluation

We evaluated the computational complexity of PAM-Tree by means of both analysis and real experiments, noting that none of the packet filter algorithms reported in the state-of-the-art section provides exact or asymptotic expressions for the average complexity. The analysis provided in this section is restricted to evaluation of the average complexity of the algorithm and does not consider

side effects such as operating system overheads, which also influence performance. Such effects are under the responsibility of the operating system and are not inherent to the algorithm being evaluated. Therefore, we will follow the usual practice in computational complexity analysis and purposely ignore such effects.

The following methodology is adopted from [4, Chapter 9] in order to derive the average complexity. The complexity of an algorithm is a function of the size of the data structure on which the operations are performed. For example, when searching or sorting a list, the data structure size is the number of elements in the list. For PAM-Tree, the data structure is the *PAM-Tree graph* defined in the previous section. First, the set of all possible PAM-Tree graphs of the same size is considered. Then, the complexity of each PAM-Tree graph is evaluated as the average number of basic operations performed per packet. Finally, the average complexity is obtained by averaging the complexities of each PAM-Tree graph over the set of all possible PAM-Tree graphs of the same size. For simplicity, the analysis will be restricted to the case of only two subfilters per filter (the case with more than two subfilters per filter is considered in [12]).

The size relates to the maximum number of basic operations that may be performed by the algorithm. For PAM-Tree, a basic operation is a visit to a subfilter node. The PAM-Tree graph size is defined accordingly.

Definition 7 (*PAM-Tree graph size*). Let us consider a packet P of m fields and a PAM-Tree graph I. Let $\{F_I(j)\}$, $1 \leqslant j \leqslant m$ be the set of all possible field j subfilters. The *size* of a PAM-Tree graph I is defined as the m-tuple $(K_1, \ldots, K_m)$ where $K_j = \#(\{F_I(j)\})$ and # represents the cardinal of a set.

Definition 8 (*Average complexity*). For each incoming packet P, let us assume, without loss of generality, that the cost of a visit to a subfilter node is equal to unity. Let us now consider the set $\Omega_{(K_1,\ldots,K_m)}$ of all possible PAM-Tree graphs of size $(K_1, \ldots, K_m)$, namely the set of all possible directed graphs (N, A) with K_i subfilters defined for packet field i. The average complexity is defined as [4, p. 77]

$$A(\Omega) = \sum_{I \in \Omega_{(K_1,\ldots,K_m)}} \tau(I) p(I) \tag{1}$$

where $\tau(I)$ is the average cost for PAM-Tree graph I and $p(I)$ its probability mass function.

Let us define the equivalence relation $\sim$ as follows: For all $I = (N, A), I' = (N', A') \in \Omega$ $I \sim I' \iff \#(A) = \#(A')$. The induced equivalence classes are formed by PAM-Tree graphs I with the same number of filters. Let us denote the equivalence class for n filters as $[n]_\Omega$. It will be assumed that all possible equivalence classes $[n]_\Omega$ have the same probability. For simplicity, it will also be assumed that at most one filter will pass the test for any given packet. Then

$$\max(K_1, K_2, \ldots, K_m) \leqslant n \leqslant \prod_{i=1}^{m} K_i, \quad m > 0. \tag{2}$$

The lower and upper bounds are obtained by considering the PAM-Tree graphs with minimum and maximum number of arcs, as shown in Fig. 5 for size (K_1, K_2). Intuitively, the fully meshed PAM-Tree graph has greater complexity, since more subfilter nodes have to be visited by incoming packets.

Let O_i be the number of possible outcomes of field i, for all packets P. For PAM-Tree graphs with subfilter nodes at two levels only (Ω_{K_1,K_2}) the average complexity is equal to

$$\begin{aligned} A(\Omega) = {} & \frac{1}{K_1 K_2 - \max(K_1, K_2) + 1} \\ & \times \sum_{n=\max(K_1,K_2)}^{K_1 K_2} \left(K_1 \left(1 + \frac{1 - K_1}{2O_1} \right) \right. \\ & + \left(\frac{K_1}{O_1} \right) \left[\frac{n - K_1}{K_1} \left(1 + \frac{1}{2O_2} \right. \right. \\ & \left. \left. \left. \times \left(\frac{(K_2 - 2)(K_1 + 1 - n)}{K_1(K_2 - 1) - 1} \right) \right) \right] \right) \end{aligned} \tag{3}$$

and, for a stream-oriented implementation with the same number of filters

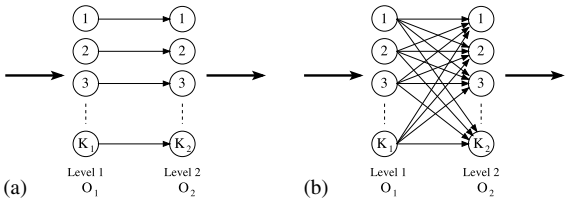

Fig. 5. Subfilter arrangement for PAM-Tree graphs of the same size: (a) minimum complexity and (b) maximum complexity.

$$A_s(\Omega) = \frac{1}{\prod_{i=1}^{m} K_i - \max(K_1, K_2, \ldots, K_m) + 1} \times \sum_{n=\max(K_1,K_2,\ldots,K_m)}^{\prod_{i=1}^{m} K_i} \frac{n+1}{2}. \quad (4)$$

The proof of both expressions is given in the Appendix A. In the above calculations, it is assumed that the processing cost per filter is the same regardless of the number of packet fields included in the filter. Actually, a significant increase in processing cost is observed when the number of bits in the filter is greater than the capacity of the ALU registers, since registers have to be fetched several times. Thus, a best case scenario for the stream-oriented architecture is considered here, since the larger the number of bits per filter, the worse the performance.

3.1. Stream-oriented architecture and PAM-Tree comparative

We performed extensive numerical simulations of the previous analytical expressions [17], and in most cases PAM-Tree outperformed the stream-oriented architecture. The percentage of cases in which PAM-Tree outperformed the stream-oriented implementation, for a system with $O_1 = O_2 = 100$ and all possible values of K_1 and K_2 in the range $1 \leqslant K_1 \leqslant O_1$ and $1 \leqslant K_2 \leqslant O_2$, is shown in Fig. 6. For a reduced number of filters (less than 80 in Fig. 6) a stream-oriented implementation is proved better. That was the case for network applications such as *tcpdump*. On the other hand, for a number of filters in the range $(105, 10000 (= O_1 O_2))$, PAM-Tree *always outperformed* the stream-oriented architecture. Note that the full

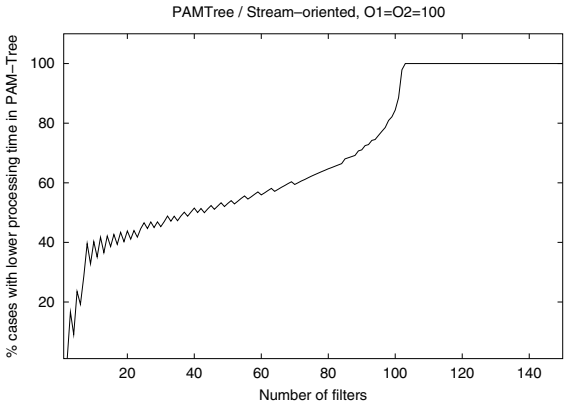

Fig. 6. Percentage of cases with lower average processing time in PAM-Tree compared to stream-oriented architecture.

range is not shown in Fig. 6 in order to provide better detail of the lower $(0, 105]$ interval, but the curve is equal to one in the range $(105, 10000]$. Traffic monitoring probes are expected to operate in the latter interval.

Fig. 7 shows the case for a traffic matrix of 256 hosts in a class B IP subnetwork. We assumed that IP was the only network protocol running in the subnetwork so that no filter at the link level was required. However, one filter per source and destination IP address pair was necessary. Thus, the resulting PAM-Tree graph has only two subfilter node levels, and the following values for K_1, O_1, K_2, O_2:

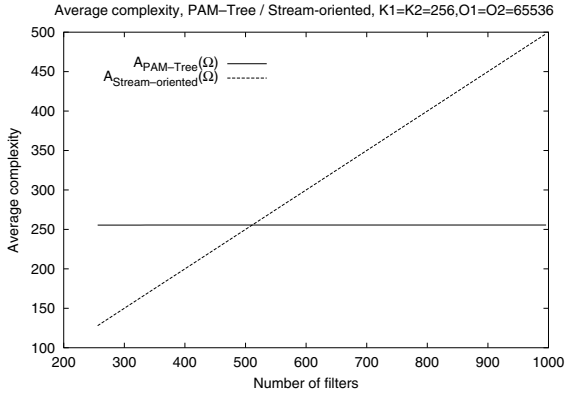

Fig. 7. Average complexity versus number of filters (n).

- Level 1—Source IP address: (4 bytes)

$$\begin{cases} O_1 = 2^{16} = 65536, & \text{Class B subnetwork} \\ K_1 = 256, & \text{256 source IP addresses} \end{cases} \tag{5}$$

- Level 2—Destination IP address: (4 bytes)

$$\begin{cases} O_2 = 2^{16} = 65536, & \text{Class B subnetwork} \\ K_2 = 256, & \text{256 destination IP addresses} \end{cases} \tag{6}$$

The results showed that in stream-oriented architecture, average complexity increased linearly with the number of filters (Eq. (4)), because filters were tested sequentially. In PAM-Tree, however, average complexity remained nearly constant, because the hierarchical arrangement of subfilter nodes reduced relative packet sojourn time. In fact, if $O_1, O_2 \to \infty$ in Eq. (3) then $A(\Omega) \to K_1$. Precisely, Fig. 7 shows a value of $A(\Omega)$ in the vicinity of $K_1 = 256$. The stream-oriented architecture is more advantageous when using fewer filters (as shown in Fig. 6), but it is less robust as the number of filters increases, as in the case of traffic monitoring.

In other words, in the stream-oriented architecture case, since average complexity is related to number of filters, we would expect a linear CPU load increase as the number of filters increases. Conversely, CPU load must remain nearly constant with PAM-Tree. Such hypotheses were verified in the experiments presented in next subsection.

3.2. Empirical evaluation

The performance figures considered in the empirical evaluation were the following:

- CPU load versus number of simultaneous filters.
- CPU load versus network load, for a fixed number of filters.
- Filter insertion and removal rate, with no pause in traffic monitoring.

In order to provide a comparative performance analysis, two different versions of traffic monitoring probes were considered, with the following filtering engines: (i) PAM-Tree and (ii) kernel-level filtering (libpcap [14] + BPF). The results obtained from the comparative performance analysis reinforce the point that packet filter techniques are not efficient for traffic monitoring applications with a large number of traffic parameters.

On the other hand, PAM-Tree runs in user space due to the availability of software to measure CPU load separately for each of the algorithm tasks (insertion, deletion, packet filter, and parameter update). Such fine grain measurements cannot be performed at kernel level [25]. Therefore, the performance figures are obtained from a worst case evaluation scenario, due to the extra overhead from packet copies between kernel and user spaces. In regards to packet filters other than BPF, either evaluation versions are unavailable (BPF+), or they only run under unavailable operating systems (Mach).

3.2.1. Performance measurement scenario

The experimental setup consisted of a network segment with three elements: (i) a traffic generator that allowed introduction of different network loads and traffic characteristics, (ii) a traffic probe running a PAM-Tree, and (iii) a traffic probe running BPF. The two probes and the traffic generator were implemented via Pentium II 350-MHz PCs in a dedicated 100 Mbps Ethernet network.

We performed the following experiments:

- Constant network load and variable number of filters: The traffic generator flooded the network with 3 Mbps constant rate traffic, and a new filter was added to both traffic probes every 10 s. That allowed enough time for the packet filter (BPF) to add the new filter, allocate the stream buffer with the traffic monitoring application and reach the steady state regime.
- Constant number of filters and variable network load: In both traffic probes, 200 filters were defined, with the network load taking on values from 500 to 95 Mbps. Regarding the traffic generator, packet headers contained random numbers and not values from real traffic traces, in

order to achieve maximum randomness. This was a worst case scenario for traffic monitoring, since a large number of non-overlapping filters had to be produced.

Figs. 8 and 9 show CPU load for both experiments. Regarding the constant load/variable filters experiment, in the packet filter implementation, CPU load increased linearly with the number of filters. We conducted an additional experiment under constant network load conditions, in which we repeatedly added the very same filter several times. The results clearly showed that the combination of libpcap and BPF did not provide any subfilter reuse whatsoever, but that was not the case for PAM-Tree. Furthermore, adding new filters to the packet filter implied extending the search space, producing a linear CPU load increase, as seen in Fig. 8.

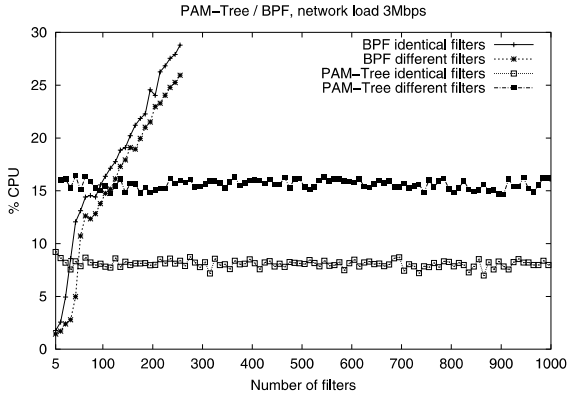

Fig. 8. CPU load versus number of filters, with constant network load of 3 Mbps.

In the case of the PAM-Tree implementation, however, we found an almost constant result, explainable by its hierarchical and reusability properties. We increased the number of filters defined in PAM-Tree up to 6000, and noted that the CPU load was under 10% in at all times. While other BPF-like systems do not have a maximum limit of 256 on the number of filters, a linear CPU load increase with the number of filters was also observed, since they also responded to the stream-oriented paradigm.

Additionally, the results in Fig. 8 were consistent with the analytical performance evaluation (see Fig. 7), which indicated a linear CPU load increase (for stream-oriented systems) versus a nearly constant CPU load increase (for PAM-Tree), with the number of filters. It must be noted that there are many contributions to CPU load, such as interruptions or packet copies. Even though such contributions are not taken into account in the analysis, the analytical model accurately predicts the behavior of both stream-oriented and PAM-Tree implementations.

Regarding the second experiment, CPU load versus network load, PAM-Tree showed a good

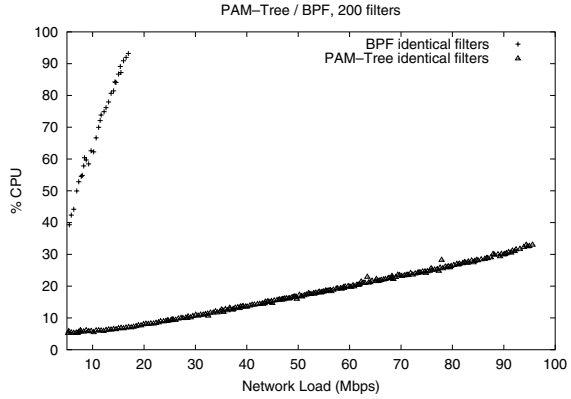

Fig. 9. CPU load versus network load, with 200 filters.

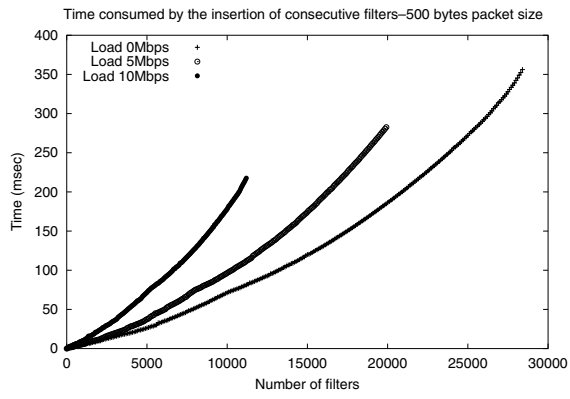

Fig. 10. Time consumed by the insertion of consecutive filters.

performance in comparison to the packet filter implementation, even though the CPU load increase was now linear for both of them.

Finally, a last experiment was performed, sending filter insertion requests to both PAM-Tree and BPF continuously at line rate. Fig. 10 shows elapsed time for PAM-Tree versus number of filters for different network loads (0, 5, and 10 Mbps). Filters were composed of four subfilters: Ethernet, IP, source and destination IP address (with different source and destination IP addresses per filter). Filter setup time with no load was on the order of 10 μs. For BPF, the insertion time per filter is in the order of milliseconds and it is constant because subfilters can not be reused. PAM-Tree is the only packet filter which can satisfy this requirement of traffic monitoring systems.

4. Implementation of a traffic monitoring system: monet

MONET is a traffic monitoring system that incorporates PAM-Tree as the filtering engine. The tool has been successfully deployed in a regional cable operator (Retena S.A., Spain) and it is currently used to monitor both cable modem headends and the company Intranet. Prior to MONET, the PROMIS tool [13] incorporated a limited version of the PAM-Tree algorithm.

MONET is a distributed system, with probes located in each network segment to be monitored and a central console that receives monitoring information. The probes have been developed using Pentium II 350 MHz PCs with Linux operating systems and Ethernet 10/100 interfaces. A graphical user interface written in Java is incorporated into the console, in order to present the information to the network manager in an intuitive way (using graphic bars, time series plots, etc.) as shown in Fig. 11(a).

The filters and parameters are also defined in the console. Fig. 11(b) shows a filter definition in the MONET console. In this example, an arbitrary number of bits in the TCP header can be selected in order to set up a filter. Then, the console translates the filter definition into (*offset*, *mask*, *value*) and this filter definition is relayed to the corresponding traffic monitoring probe that is running the PAM-Tree algorithm.

5. Conclusions

We have presented a novel PAM-Tree algorithm for traffic monitoring in this paper. We have shown that stream-oriented algorithms (packet filters) are not suitable for monitoring scenarios where thousands of filters run concurrently and dynamically change. In fact, neither the memory management scheme (using stream buffers) nor the filtering technique (implementing packet streams) are tailored to the special case of network traffic monitoring. We believe that performance comparisons with newer versions of BPF, such as

(a) (b)

Fig. 11. MONET console: (a) console snapshot and (b) filter definition interface.

BPF+ (still not publicly available) will yield similar results. Even though these newer algorithms promise to avoid filter redundancy, they still provide one packet stream per parameter.

In contrast, PAM-Tree provides very good performance for traffic monitoring systems, and is especially effective at allowing filter insertion and removal with no service interruption. In addition, we are currently porting PAM-Tree to embedded Linux in order to develop traffic measurement probes for ATM and IP-over-WDM networks. Concerning future work, we plan to extend PAM-Tree's capabilities to support not only packet filtering but also connection filtering. By adding the TCP connection state to the algorithm functionalities, we will be able to provide TCP connection statistics, such as duration, bytes transferred, and more.

Acknowledgements

The authors would like to thank our anonymous reviewers for their helpful comments, which resulted in significant improvements to the quality of this paper.

Appendix A

A.1. PAM-Tree pseudocode

A brief pseudocode for the most important PAM-Tree primitives is provided in this appendix. The full PAM-Tree specification is written with the input–output automata formalism [12]. The PAM-Tree actions are displayed in Fig. A.1.

Given a PAM-Tree graph $G = (N, A)$:

- Symbol // indicates comments.
- For a parameter Q, the filter $Q.F$ is composed by the subfilters $F_1, \ldots, F_N$.
- subfilter(subfilter node n) $= F_j$ if and only if the subfilter node n represents the subfilter F_j.
- sons(subfilter node n) $= \{\{n_1, n_2, \ldots, n_m\} \in N$ such that $\{(n, n_1), (n, n_2), \ldots, (n, n_m)\} \in A\}$.
- son(subfilter node n, subfilter G_k) $= \{n_i \in N$ such that $(n, n_i) \in A$ and subfilter $(n_i) = G_k\}$.
- parameters(subfilter node n) $= \{\{Q_1, Q_2, \ldots, Q_r\} \in n$, where $Q_i = (\mathrm{id}, F, \mathrm{UpdateFunction})\}$.

SetParam(Parameter Q)

preconditions: PAM-Tree graph $G = (N, A)$, Parameter $Q = (\mathrm{id}, F, \mathrm{UpdateFunction})$, subfilter node CurrentNode $= \emptyset$

effects:

```
// Current node is root node
CurrentNode ← root(G.N)
// For each subfilter in the filter associated to the parameter
FOREACH (F_i ∈ Q.F)
   // If the subfilter is not already in the graph, add the subfilter
   IF F_i ∉ sons(CurrentNode) THEN
      sons(CurrentNode) ← sons(CurrentNode) ∪ F_i
   ENDIF
   // Proceed with the next level in the tree
   CurrentNode ← son(CurrentNode, F_i)
ENDFOREACH
// Attach the parameter to the last subfilter node in the filter Q.F
parameters(CurrentNode) ← parameters (CurrentNode) ∪ Q
```

DelParam(Parameter Q)
preconditions: PAM-Tree graph $G = (N, A)$, Parameter $Q = (\mathrm{id}, F, \mathrm{UpdateFunction})$,
subfilter node CurrentNode $= \emptyset$, subfilter node NextNode $= \emptyset$
effects:
// Current node is root node
CurrentNode $\leftarrow$ root$(G.N)$
// For each subfilter in the filter associated to the traffic parameter
FOREACH $(F_i \in Q.F)$
NextNode $\leftarrow$ son(CurrentNode, F_i)
// If the subfilter is not a part of a filter associated to another parameter,
// remove the subfilter from the PAM-Tree graph.
IF (NextNode $\notin \{Q_i.F : \forall Q_i \in$ parameters(n) where $n \in N\}$) THEN
sons(CurrentNode) $\leftarrow$ sons(CurrentNode) $- F_i$
ENDIF
// Proceed with the next node down in the hierarchy
CurrentNode $\leftarrow$ NextNode
ENDFOREACH
// Remove the parameter
parameters(CurrentNode) $\leftarrow$ parameters(CurrentNode) $- Q$

Packet(Packet P)
preconditions: PAM-Tree graph $G = (N, A)$, Packet P, subfilter node CurrentNode $= \emptyset$,
subfilter node Sons $= \emptyset$
effects:
// Current node is root node
CurrentNode $\leftarrow$ root$(G.N)$
// Sons stores all nodes to check
Sons $\leftarrow$ sons(CurrentNode)
// Check if the packet verifies node subfilters in Sons
FOREACH (CurrentNode $\in$ Sons)
IF (CurrentNode verified by P)
// Update all parameters attached to the subfilter node using the
// parameter update function (Definition 6)
UpdateFunction$(\{Q_i : Q_i \in$ parameters(CurrentNode)$\})$
// Keep searching down in the corresponding branch
Sons $\leftarrow$ Sons $\cup$ sons(CurrentNode)
ENDIF
// Delete current subfilter node reference from Sons to check the next one
Sons $\leftarrow$ Sons $-$ CurrentNode
ENDFOREACH

The SetParam() action is enabled when the PAM-Tree receives a parameter set request. By execution of this action a parameter is inserted in the structure. That is, the structure is updated with a set of subfilter nodes, if necessary, and a parameter node. The process starts from the root node. For each level, the subfilter being inserted is compared with previously inserted subfilter nodes. If it already exists, the subfilter node is reused for this new parameter; otherwise a new

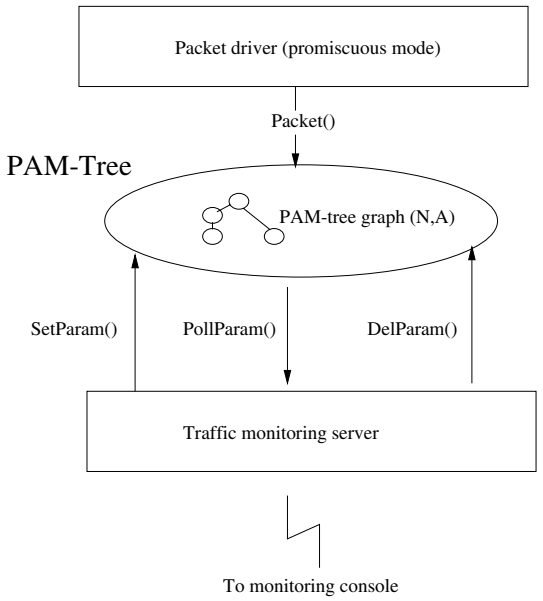

Fig. A.1. PAM-Tree interface.

subfilter node is inserted. The process is repeated until the last subfilter node is reached. Finally, the parameter is attached to the last subfilter node.

Traffic parameters can be eliminated by execution of the action DelParam(). From the root node, the corresponding set of subfilters and parameter node are eliminated if and only if such nodes are not used to evaluate any other monitoring parameter.

The Packet() action is the core of the algorithm. For each packet, the function performs a test on a set of subfilters starting from the root node of the PAM-Tree graph. If the packet satisfies several filters, the associated parameters are updated. The processing cost to update the whole set of parameters is reduced since a test for a subfilter allows updating more than one parameter.

Finally, the action PollParam() is executed in order to look for the value of a parameter. By execution of this action the traffic server gets the value of the required traffic parameter.

A.2. Proof of Eq. (3) (PAM-Tree with two subfilter levels (Ω_{K_1,K_2}))

The starting point is the definition of average complexity (Eq. (1)). Let the random variable M denote the walk length. Let the random variable X denote the cost, measured in number of subfilter nodes visited. Let X_i be the number of level i subfilter nodes visited, $i = 1, 2$. Clearly $X = X_1 + X_2$ and, necessarily, $M \in \{1, 2\}$ thus

$$A(\Omega) = P(M = 1)E[X_1 + X_2|M = 1] + (1 - P(M = 1))E[X_1 + X_2|M = 2] \quad (A.1)$$

If the walk length M is equal to 1, then $X_2 = 0$ and $X_1 = K_1$ a.s., since all level 1 subfilters must have been visited by the incoming packet and none of them is verified. On the contrary, if $M = 2$, and assuming that all level 1 subfilter nodes have the same probability of being visited, then $E[X_1] = (1/K_1)\sum_{i=1}^{K_1} i = (K_1 + 1)/2$. In order to calculate $P(M = 1)$ let O_i be the number of possible outcomes of field i, for all packets P. Then $P(M = 2) = K_1/O_1$ and, using (A.1),

$$A(\Omega) = K_1\left(1 + \frac{1 - K_1}{2O_1}\right) + \left(\frac{K_1}{O_1}\right)E[X_2] \quad (A.2)$$

where $E[X_2]$ depends on each particular PAM-Tree graph. For instance, if a level 1 subfilter node is verified in the left PAM-Tree graph shown in Fig. 5, then only one subfilter node at level 2 will be visited and $X_2 = 1$ a.s. For the right graph, however, X_2 is a uniform random variable $U(1, K_2)$ and, thus, $E[X_2] = (K_2 + 1)/2$. Since the equivalence classes $[n]_\Omega$ (Eq. (2)) define a partition of Ω then

$$E[X_2] = E\left[E[X_2|[n]_\Omega]\right] \quad (A.3)$$

For any PAM-Tree graph $I = (N, A)$ let $a_i = \mathrm{card}(\{(n_k, n_l) \in A|k = i, l = 1, \ldots K_2\})$, $i = 1, \ldots, K_1$ be the number of arcs departing from node i. Fig. A.2 shows an example with $a_i = 4$. Now, for any PAM-Tree graph $I \in [n]_\Omega$, the tuple $(a_1, \ldots, a_{K_1})$ is a random vector that fulfills:

$$a_1 + a_2 + \cdots + a_{K_1} = n, \quad (A.4)$$

$$a_i \in [\max(1, (n - (K_1 - 1)K_2)), \min(K_2, n - (K_1 - 1))], \quad 1 \leqslant i \leqslant K_1. \quad (A.5)$$

Eq. (A.4) follows from the fact that $I \in [n]_\Omega$; Thus, the total number of filters is equal to n. For two subfilter levels the number of filters is equal to the number of arcs in the graph I. Eq. (A.5) provides the range of each a_i, noting that the lower bound is

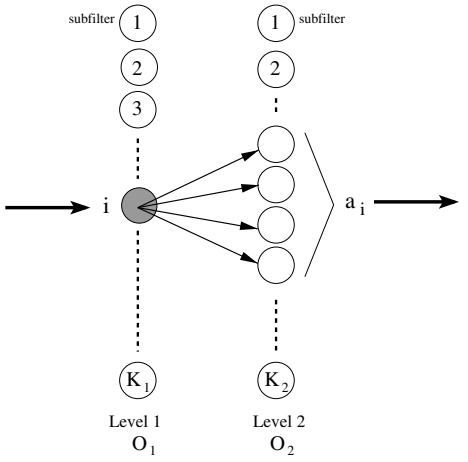

Fig. A.2. Example of subfilter arrangements.

$n-(K_1-1)K_2$ if $n>(K_1-1)K_2$ and 1 otherwise. The upper bound cannot be larger than K_2 in any case. Since all level 1 subfilters nodes must have at least one departing arc, the upper bound is always lower than $n-(K_1-1)$. On the other hand, assuming that level 1 subfilter nodes have the same probability of being visited, $E[x_2|[n]_\Omega]$ can be found by conditioning to any subfilter node i and

$$\begin{aligned}E[X_2|[n]_\Omega] &= E\big[E[(X_2|[n]_\Omega)|a_i=j]\big]\\ &= \sum_{j=\max(1,(n-(K_1-1)K_2))}^{\min(K_2,n-(K_1-1))} E[(x_2|[n]_\Omega)|a_i=j]P(a_i=j).\end{aligned} \tag{A.6}$$

For all PAM-Tree graphs $I\in[n]_\Omega$, the probability mass function for a_i is equal to

$$P(a_i=j)=\frac{\binom{K_2-1}{j-1}\binom{(K_1-1)(K_2-1)}{n-K_1-(j-1)}}{\binom{K_1(K_2-1)}{n-K_1}} \tag{A.7}$$

since at each level one subfilter node has at least one departing arc and the remaining $n-K_1$ arcs may be placed in

$$\binom{K_1(K_2-1)}{n-K_1}$$

different manners, out of which only

$$\binom{K_2-1}{j-1}\binom{(K_1-1)(K_2-1)}{n-K_1-(j-1)}$$

provide j departing arcs from subfilter node i. On the other hand, let the probability of passing a filter at level 2, assuming that node i was visited at level 1, be equal to a_i/O_2. Following the same argument as with Eq. (A.2)

$$E[(X_2|[n]_\Omega)|a_i]=a_i\left(1+\frac{1-a_i}{2O_2}\right) \tag{A.8}$$

since $(X_2|[n]_\Omega)$ is equal to a_i a.s. if none of the a_i subfilter nodes at level 2 is verified and equal to

$$\frac{1}{a_i}\sum_{j=1}^{a_i} j=\frac{a_i+1}{2}$$

otherwise. Then

$$\begin{aligned}&E[X_2|[n]_\Omega]\\ &\quad=\sum_{j=\max(1,(n-(K_1-1)K_2))}^{\min(K_2,n-(K_1-1))} j\left(1+\frac{1-j}{2O_2}\right)P(a_i=j)\\ &\quad=E[a_i]+\frac{1}{2O_2}(E[a_i]-E[a_i^2])\\ &\quad=\frac{n-K_1}{K_1}\left(1+\frac{1}{2O_2}\left(\frac{(K_2-2)(K_1+1-n)}{K_1(K_2-1)-1}\right)\right).\end{aligned} \tag{A.9}$$

In the derivation of the previous expression it must be noted that a_i is a hypergeometric random variable. Now, substitute the previous expression in Eq. (A.3), and Eq. (A.3) in Eq. (A.2) to obtain

$$\begin{aligned}A(\Omega)=&\frac{1}{K_1K_2-\max(K_1,K_2)+1}\\ &\times\sum_{n=\max(K_1,K_2)}^{K_1K_2}\left(K_1\left(1+\frac{1-K_1}{2O_1}\right)\right.\\ &+\left(\frac{K_1}{O_1}\right)\left[\frac{n-K_1}{K_1}\left(1+\frac{1}{2O_2}\right.\right.\\ &\left.\left.\left.\times\left(\frac{(K_2-2)(K_1+1-n)}{K_1(K_2-1)-1}\right)\right)\right]\right)\end{aligned} \tag{A.10}$$

which is precisely Eq. (3). The above analysis can be extended to any size $(K_1,\ldots,K_m)$ (Definition 7) using an induction argument for subsequent levels

and taking into account the dependencies between subfilters at different levels [12].

A.3. Proof of Eq. (4) (stream-oriented architecture)

From Section 1.2.1 recall that the stream-oriented implementation is based on sequential testing of filters. Furthermore, it will be assumed that only one filter will pass the test for any given packet. For a fixed number of filters, the average complexity is thus equal to the average complexity of a linear search. Since a graph I in the equivalence class $[n]_\Omega$ defines n filters, then for all graphs $J \in [n]_\Omega$,

$$\tau(J) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}$$

and the complexity $A_s(\Omega)$ can be derived as follows (see [4, p. 77]):

$$\begin{aligned} A_s(\Omega) &= \sum_{I \in \Omega_{(K_1,\ldots,K_m)}} \tau(I)p(I) \\ &= \sum_{\forall [n]_\Omega} \frac{n+1}{2} p([n]_\Omega) \end{aligned} \tag{A.11}$$

and, from Eq. (2),

$$\begin{aligned} A_s(\Omega) = {} & \frac{1}{\prod_{i=1}^{m} K_i - \max(K_1, K_2, \ldots, K_m) + 1} \\ & \times \sum_{n=\max(K_1,K_2,\ldots,K_m)}^{\prod_{i=1}^{m} K_i} \frac{n+1}{2} \end{aligned} \tag{A.12}$$

which is Eq. (4).

References

[1] K.R. Apt, J. Brunekreef, V. Partington, A. Schaerf, Alma-0: An imperative language that supports declarative programming, ACM Transactions on Programming Languages and Systems 5 (20) (1998) 1014–1066.

[2] M.L. Bailey, B. Gopal, M.A. Pagels, L.L. Peterson, PathFinder: A pattern-based packet classifier, in: Proceedings of the First Symposium on Operating Systems Design and Implementation, Usenix Association, November 1994.

[3] A. Begel, S. McCanne, S.L. Graham, BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture, in: Proceedings of ACM SIGCOMM'99, Cambridge, MA, September 1999.

[4] K. Berman, J. Paul, Fundamentals of Sequential and Parallel Algorithms, PWS Publishing Company, Boston, 1997.

[5] D.R. Engler, M.F. Kaashoek, DPF: Fast, flexible message demultiplexing using dynamic code generation, in: Proceedings of ACM SIGCOMM Symposium on Communications Architectures and Protocols, Standford, CA, August 1996.

[6] V. Fuller, T. Li, J. Yu, K. Varadhan, Classless inter-domain routing (CIDR): An address assignment and aggregation strategy, RFC 1519, Internet Engineering Task Force, September 1993.

[7] P. Gupta, N. McKcown, Packet classification on multiple fields, in: Proceedings of ACM SIGCOMM'99, Cambridge, MA, August 1999, pp. 147–160.

[8] IANA. Assigned Internet Protocol Numbers. http://www.iana.org/assignments/protocol-numbers.

[9] V. Jacobson, C. Leres, S. McCanne, The tcpdump manual, University of California, Berkeley, June 1997.

[10] T.V. Lakshman, D. Stiliadis, High-speed policy-based packet forwarding using efficient multi-dimensional range matching, in: Proceedings of ACM SIGCOMM'98, Vancouver, BC, September 1998, pp. 203–214.

[11] N.A. Lynch, M.R. Tuttle, An introduction to input/output automata, CWI Quarterly, September 1989.

[12] E. Magaña, Efficient filtering techniques for traffic monitoring in communication networks, Ph.D. Thesis, Public University of Navarra, Pamplona, Spain, 2001 (in Spanish).

[13] E. Magaña, J. Aracil, J. Villadangos, PROMIS: A reliable real-time network management tool for wide area networks, in: Proceedings of IEEE Euromicro 98, Volume II, Vasteras, Sweden, August 1998.

[14] S. McCanne, C. Leres, V. Jacobson, Libpcap 0.4, June 1997. ftp://ftp.ee.lbl.gov/libpcap.tar.Z.

[15] S. McCanne, V. Jacobson, The BSD packet filter: A new architecture for user-level packet capture, in: Proceedings of the 1993 Winter USENIX Conference, San Diego, CA, January 1993, pp. 259–269.

[16] J.C. Mogul, R.F. Rashid, M.J. Accetta, The packet filter: An efficient mechanism for user-level network code, in: Proceedings of ACM Symposium on Operating Systems Principles, Austin, TX, November 1987, pp. 39–51.

[17] D.C. Montgomery, G.C. Runger, Applied Statistics and Probability for Engineers, second ed., Wiley, New York, 1999.

[18] G.R. Ryan, The new generation of network monitoring systems, ATG's Communications & Networking Technology Guide Series, 1997.

[19] Silicon Graphics, IRIX Device Driver Programmer's Guide, IRIX 6.5.3., 1999.

[20] V. Srinivasan, G. Varghese, S. Suri, M. Waldvogel, Fast and scalable layer four switching, in: Proceedings of ACM SIGCOMM'98, Vancouver, BC, September 1998, pp. 191–202.

[21] W. Stallings, SNMP, SNMPv2, SNMPv3 and RMON 1 and 2, third ed., Addison-Wesley, Reading, MA, 1999.

[22] R. Subramanyan, J.M. Alonso, J.A.B. Fortes, A scalable SNMP-based distributed monitoring system for heterogeneous network computing, in: Proceedings of Supercomputing 2000, Santa Fe, NM, May 2000.
[23] UNIX International OSI Work Group Revision, Data link provider interface specification, Revision 2.0.0 edition, August 1991.
[24] S. Waldbusser, Remote network monitoring management information base version 2 using SMI v2, Internet Engineering Task Force, RFC 2021, January 1997.
[25] K. Yaghmour, M.R. Dagenais, Measuring and characterizing system behavior using kernel-level event logging, in: Proceedings of the 2000 USENIX Annual Technical Conference, June 2000.
[26] M. Yuhara, B.N. Bershad, C. Maeda, J.E.B. Moss, Efficient packet demultiplexing for multiple endpoints and large messages, in: Proceedings of the 1994 Winter USENIX Conference, San Francisco, CA, January 1994, pp. 153–165.

Eduardo Magaña received his M.Sc. and Ph.D. degrees in Telecommunications Engineering from Public University of Navarra, Pamplona, Spain, in 1998 and 2001 respectively. During 2002 he was a postdoctoral visiting research fellow at the Department of Electrical Engineering and Computer Science, University of California, Berkeley. He is currently an assistant lecturer at Public University of Navarra. His main research interests are network monitoring, multimedia services and wireless networks.

Javier Aracil received the M.Sc. and Ph.D. degrees (Honors) from Technical University of Madrid in 1993 and 1995, both in Telecommunications Engineering. In 1995 he was awarded with a Fulbright scholarship and was appointed as a Postdoctoral Researcher of the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. In 1998 he was a research scholar at the Center for Advanced Telecommunications, Systems and Services of The University of Texas at Dallas and he is currently a tenured Associate Professor at Public University of Navarra, Spain. His research interest are in Internet services analysis and performance evaluation of communication networks. Dr. Aracil is a member of the editorial board of SPIE/Kluwer Optical Networks Magazine.

Jesús Villadangos is an associate professor at the Public University of Navarra (UPNA) in Spain. He receives his B.S. degree in Physics from the Universidad del País Vasco and his Ph.D. in Communications Engineering from Public University of Navarra in 1999. He has been a research assistant at the Universität Hannover and Münster in Germany. Since 2000 he has been jointly in Telematics Engineering at the UPNA. His research interests include distributed algorithms, adaptive network control and performance evaluation of communication networks and distributed algorithms.